INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1067

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

AN OBJECT-ORIENTED MODEL IN THE CONCURRENT LOGIC PROGRAMMING LANGUAGE PARLOG

Antoine RIZK
Jean-Marc FELLOUS
Michel TUENI

Août 1989

RR_1067

Programme 1

An Object-Oriented Model in the Concurrent Logic Programming Language PARLOG

Un Modèle Orienté-Objet au dessus du Langage Logique Parallèle PARLOG

Antoine RIZK‡ - Jean-Marc FELLOUS† - Michel TUENI†

Abstract

This paper describes an object-oriented model and its realisation in the concurrent logic programming language PARLOG. The model benefits directly from the fact that it unifies the concepts of logic programming, object-oriented programming and concurrent programming languages. It also differs substantially from other models both in its conception as well as in its implementation.

Keywords

Logic Programming, Concurrent Programming, Object-Oriented, PARLOG.

Résumé

Ce papier décrit un modèle orienté-objet au dessus d'un langage de logique parallèle : PARLOG.

Combinant deux formalismes puissants de programmation, à savoir la logique et le parallélisme, PARLOG est dans la ligne des langages de logique parallèle tels que "Concurrent Prolog" et "Guarded Horn Clauses". Au-dessus de PARLOG, un modèle Orienté-Objet a été conçu et implémenté. Il est caractérisé par une uniformisation des différents concepts autour d'une entité de base : l'objet. En particulier, tous les objets sont instances d'un unique Méta-Objet et sont caractérisés par leurs comportements. Deux modes de hiérarchie avec des mécanismes de délégation sont supportés : "part-of" et "is-a". Le modèle objet préserve toutes les fonctionnalités du paradigme sous-jacent, à savoir PARLOG. Enfin, un exemple illustrera le modèle proposé.

† BULL — MTS, 7 rue Ampère, 91343 MASSY Cedex (FRANCE), ☎ [33] (1) 64.47.84.67.
‡ INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (FRANCE), ☎ [33] (1) 39.63.54.35, telex 697 033 F, email: rizk@minos.inria.fr.

An Object-Oriented Model in the Concurrent Logic Programming Language PARLOG

1.Introduction

The work described in this paper was motivated by two principal and complementary requirements : the *design* of a sophisticated object-oriented model that can benefit from the power of concurrent logic languages, and the *realisation* of the model in such a way as :

i) to render it more natural for the representation of concepts at different levels of abstraction ;

ii) to provide control structures that are needed in many applications and are rarely provided in common object-oriented models.

The model is designed in such a way that it has been easily implemented in the form of a simple preprocessor language on top of PARLOG and could be implemented graphically through an interactive programming environment. The paper starts by introducing the language PARLOG and the concepts that have strongly inspired our work. It then proceeds to present the model we developed. Finally, we illustrate the use of the model through a small simulation example and describe how this example is realised in PARLOG.

2. Background

In this section we present the language PARLOG and the concepts that have strongly inspired the work described in this paper. A thorough understanding of the details of the language and the concepts described is not essential for an appreciation of the model we introduce. It is however necessary for an understanding of the implementation.

2.1 PARLOG

Intrinsic to the resolution procedure that governs the execution of logic programs, there are two main areas of non-determinism giving the potential for concurrent interpretation : firstly, in the selection of the predicate call in the body of a clause and secondly in the selection of the unifying clause from the set of clauses constituting the corresponding predicate. The two forms of parallelism resulting from these interpretations are termed AND and OR respectively [1].

PARLOG [2] is a concurrent programming language that features AND parallelism with stream inter-process communication and OR parallelism with don't care non-determinism using guarded Horn clauses. A program in PARLOG is a finite set of clauses of the form :

H <- G1,..,Gn : B1, .., Bm.

The declarative semantics of this clause is as in Prolog, that is, H is true if G1 and .. Gn and B1 and .. Bm are true. The operational semantics is however quite different. The ':' is the *commit* operator and is the parallel counterpart of the Prolog *cut*. A clause is a candidate for selection if the arguments in the head unify and also the guard succeeds. Selection amongst candidate clauses is effected at random, hence the don't care non-determinism as opposed to the don't know non-determinism of Prolog.

As well as the declarative and the procedural reading of logic programs, programs written in PARLOG exhibit a *process* interpretation. A process is a predicate call, and communication between processes is effected through the instantiation of variables common to such calls. Synchronization between processes and the direction of communication is governed by the explicit declaration of the argument *modes*, input (?) or output (^), for each predicate. The declaration of modes reduces much of the unification process to compilable filtering/pattern-matching and establishes directionality in the communication channels. Synchronization of processes is achieved by suspension on input variables until they are instantiated.

One useful feature of PARLOG is the provision of parallel, sequential and neutral conjunction operators which specify that relation calls or search for candidate clauses is to be effected in parallel, serially, or at the implementer's will respectively. The example below shows the specification of the deterministic "append" operation. In this case, search for the candidate clause is specified sequential by the operator ';' :

```
mode append (?,?,^).
append ([Head|A], B, [Head|C]) <- append (A,B,C) ;
append ([], B, B).
```

2.2 Objects in PARLOG

PARLOG is the latest of a family of similar languages that derives from the Relational Language [3] and that includes Concurrent Prolog [4] and Guarded Horn Clauses [5]. A major advantage of these languages is that they lend themselves naturally to various models of programming, such as the communicating sequential processes and the object oriented. The object oriented model was first introduced by [6] and first formalised in Vulcan [7] on top of Concurrent Prolog, this then led to the development of similar languages on top of GHC[8] and PARLOG[9]. Objects in these languages are represented and instantiated as perpetual processes, and inheritance is performed by filtering and delegation. A perpetual process is represented as a recursive predicate that reduces itself to itself through recursion and that contains its local state in its non-shared arguments. Predicate arguments that are shared amongst other predicates in the same goal serve for communication amongst objects.

We illustrate this here with two template objects "object1" and "object2" with object2 delegating unknown messages to object1 (variables are identified with capital letters) :

```
mode object1 (?, ?).
object1 ([method11(Parameters)|Moremethods], Localstate) <-
        body-method11(Parameters,Localstate,Newstate), object1(Moremethods, Newstate).
object1 ([method12(Parameters)|Moremethods], Localstate) <-
        body-method12(Parameters,Localstate,Newstate), object1(Moremethods, Newstate).

mode object2 (?, ?).
object2 ([method21(Parameters)|Moremethods], Localstate) <-
        body-method21(Parameters,Localstate,Newstate), object2(Moremethods, Newstate).
.
.
object2 ([Unknownmethod|Moremethods], Localstate) <-
        object1(Unknownmethod, ...) , object2 (Moremethods, Localstate).
```

Instantiation of these objects can be effected through the goal :

```
<-object1(Messagestream1,initstate1),object2(Messagestream2,initstate2),
        object3(Messagestream1, Messagestream2).
```

Where object3 has mode (^,^) and generates two streams of messages for consumption by object1 and object2.

Although the concept of objects in our model is inspired directly from the above principle, its representation differs from the general template shown here as will be described in section 3.6. It also caters for the dynamic modification of objects and the representation of the Activity Network and the Object Manager concepts (sections 3.2 and 3.4).

3. The Model

This section describes the main concepts around which the model is built and outlines its implementation. We first describe objects and their basic constituents, followed by a more detailed description of the novel concepts introduced. We then demonstrate these concepts through a short typical simulation application that shows the

advantages of the model presented. Finally, we describe the implementation of the model using the example application for illustration.

3.1 The Object

Like all of the known object-oriented models, the basic entity in our model is the object. Information processing in each object is event driven in the sense that an object responds to its environment through the reception of messages. All objects are exactly of a single type and are considered to be instances of a single meta object that is unique in the whole system and visible only to the implementation. Instance variables of the meta object correspond to the basic components of an object and are illustrated in figure 1. Since the meta object is mostly of interest to the implementation, we defer its presentation to section 3.6. In what follows, we describe briefly each of an object's basic components.

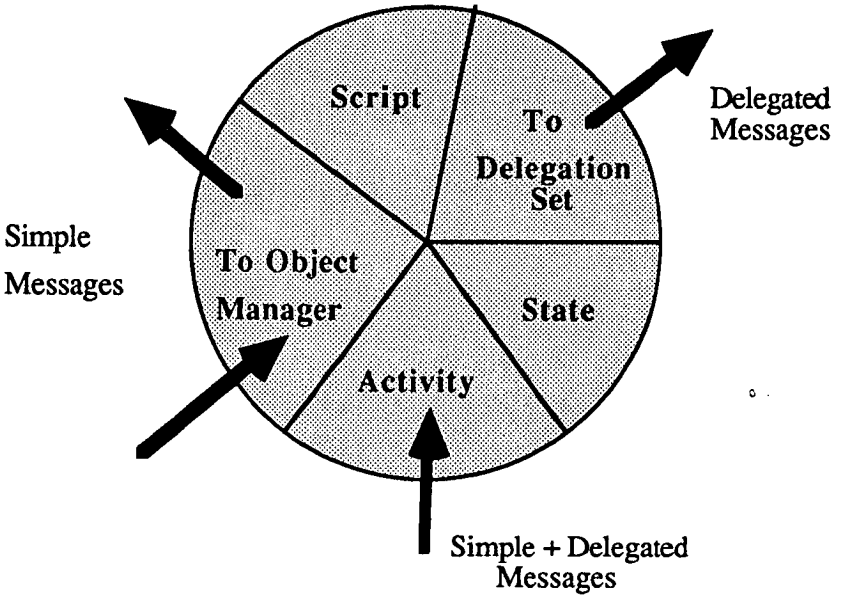

Figure 1. An Object and its Message Streams

3.1.1 The State Set

Each object possesses a state set that characterises it and represents the data local to each object. Each state element consists of a couple attribute/values which can only be modified by the object itself. The characteristics of a state set is that it is completely dynamic in the sense that not only values can be modified but also elements can be added or removed. For example, the state [(colour (blue)), (owner (rizk)), type((make (renault)), model (5))] can at any time have another element added to it such as (horsepower (4)), colour (blue, light) etc ...

3.1.2 The Script Set

The script set is represented as a set of couples each consisting of :
- a message name ;
- a name of a predicate (method) that treats the message.

As with the state set, the script set is entirely dynamic. New scripts can be added and old ones modified.

3.1.3 The Delegation Set

In the general case, an object possesses a neighbouring environment that we call the delegation set or is-a set. This set comprises those objects to which unmatched messages are passed. Unmatched messages are forwarded *simultaneously* to all the objects in the delegation set. This allows on one hand, to distribute the processing of a message into a set of objects, and, on the other hand, for delegated objects to respond differently and in a complementary manner to the same message. It however necessitates that responses to messages be coherent.

The delegation set is considered as an invariant state element of each object and is initialised to empty at object creation time. Two other invariant state elements are also created with each object to help the implementation in the buffering of messages. They are initialised to empty at creation time and updated dynamically to contain the number of messages waiting to be consumed by the object and the object's maximum buffer size (sections 3.3 and 3.6.1).

3.1.4 The Object Manager Link

Each object in a delegation set possesses a link in a control/part-of tree hierarchy to a parent object called the *Object Manager*. An Object Manager is responsible for the creation and control of an object network. Contrary to what this nomenclature may lead to believe, an Object Manager is not a special kind of object, it is an ordinary object that has acquired *dynamically* a set of system methods with certain control priviliges. These will be described in section 3.2.

3.1.5 The Activity Network

Whenever an object is not responding to an external message it may process a private suite of messages called activity. Activities are specified through an *activity network* which is a finite or an infinite set of messages, connected through a hierarchical structure reflecting the serialisation or the parallelism required in their processing (section 3.4).

The presence of activities means that objects are *really* active in the sense that they are not simply suspended whilst waiting for messages from external objects.

At any time, an object can be asked, by its Object Manager
- to indefinitely suspend its activity, i.e become passive ;
- if it is passive, to be attributed an activity ;
- to have its activity network changed dynamically ;
- to interrupt its activity and execute a forwarded message.

3.1.6 The invariant method set

Every object without exception possesses at least two methods in its script set : the termination method and the delegation method. The termination method allows for an object to execute its 'last wishes' and to liberate its links and resources after an order of termination. This order can only come from/via its Object Manager. The delegation method in its simplest form reforwards any unknown message simultaneously to all objects in the delegation set. The termination and delegation methods can be replaced if needed, for e.g. by more verbose versions for debugging purposes.

3.2 The Object Manager

Objects are organised around groups, each group being under the control of an Object Manager. The Object Manager is an ordinary object in every respect except for the presence of additional system methods to its script namely :

i) the methods necessary for the creation and instantiation of objects ;
ii) the methods necessary for the management of objects in the group.

The latter kind of methods includes :
- methods for resource management. For example, messages are sent via a bounded buffer in order to allow for objects to be eager. The size of the buffer can be modified dynamically at anytime by sending an appropriate message to the group Object Manager. Moreover, the Object Manager can be asked to control the fairness of message reception by indicating the proportion of messages from each object to be accepted for inclusion in the message buffer ;
- methods allowing for objects to communicate, without going through the delegation network, with other objects in the same or a different group in the application ;
- user defined methods which allow the verification of the general consistency of the group.

Other than its role as a controller, the Object Manager behaves as a representative of its group and can be considered as an encapsulation, through a part-of relationship, of the group component objects. An external object would then address itself to the Object Manager without needing to know the component objects or even their nature.

We can note at this stage, that we can exhibit a different type of hierarchy, other than the is-a one, linking each object (whether Manager or not) to a unique Object Manager. This hierarchy has the properties of being *both* a *control hierarchy* as well as a *part-of hierarchy*.

3.3 Messages and methods

Messages constitute the basic element for communication amongst objects. They circulate in an asynchronous, buffered, bidirectional manner and in packets. The buffering is performed on a blocking send basis ; both the specification of the buffer size as well as the blocking behaviour are determined by the Object Manager.
We distinguish two kinds of messages :
- *simple messages* : they are sent explicitly across the part-of/control hierarchy
- *delegated message* : they broadcast implictly in the 'is-a' hierarchy.

Both of these messages consist of a generic name and a set of parameters.

Message parameter passing is done through unification. In this sense, parameters can be either constants, bound variables, or unbound variables. In the latter case, the receiver, if it requires the variable, will suspend until the variable is instantiated by the transmitter, otherwise it may use the variable as a mailbox for back communication.

The sender can also specify a delegated object to which a reply is to be sent. The receiver in this case can make use of this information for the treatment of the message. In addition, the sender can specify a *complaint object* for handling a failed message.

3.4 The Object Activity

An activity is a series of messages (finite or infinite) to be consumed by the object. It is divisible into subactivities which are organised as a graph or network reflecting the generation of these messages.
From a general point of vue, an activity attached to an object represents the behaviour of the object in the absence of external messages. The subactivities are treated by the object either in sequence, or in parallel, or in a combination of the two through a controllable biased merge. Control of the biased merge is done by the Object Manager and is specified as a proportion x:y where x is the number of messages to be selected from a given subactivity out of every y number of messages.
The subactivity graph may have one or more branches issuing from the same *test node* , which allow for the treatment of certain subactivities depending on some condition relating to the object's state or the behaviour of other objects.
Consider, for example, the object "alarm clock" shown in figure 2. Its activity can be decomposed into two parallel subactivities :
- maintaining the hour (its state "h/mn/s") ;
- ringing the bell when it is 8 o'clock.

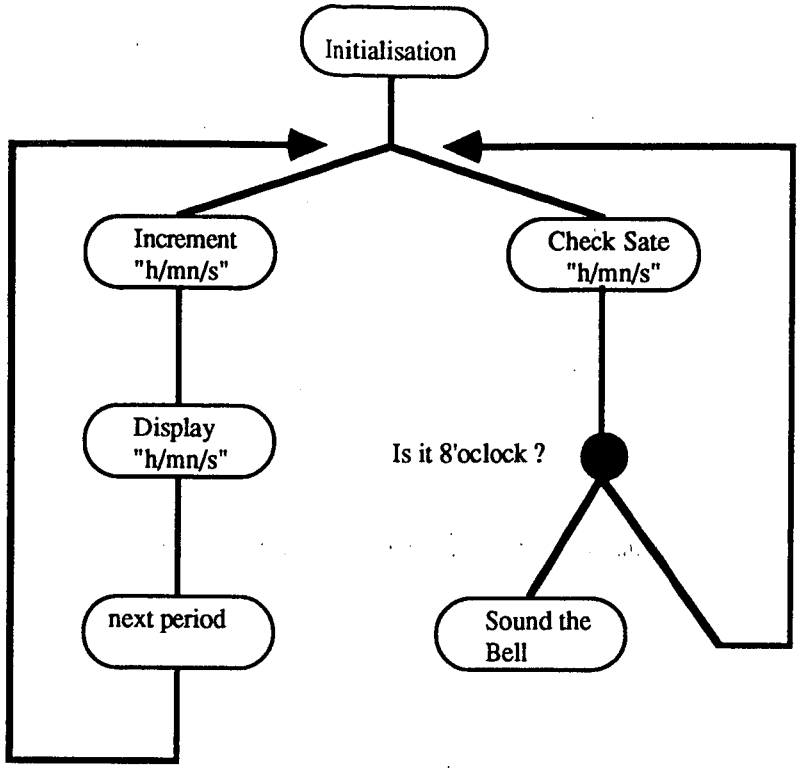

Figure 2 . Activity of an Alarm Clock

We note the following points :

- the activity of alarm clock possesses sequential and parallel subactivities as well as a test ;
- it consists of an infinite series of messages due to the presence of loops ;
- at any moment, its activity can be interrupted by the arrival of a message from an object wanting to know the time ;
- the merge proportion can be specified in a way that depends on the resolution of the clock as well as the error allowed in the time of sounding the bell.

3.5 Object Organisation

In this section, we will illustrate through a small example, the application of the concepts and methods described so far. The example we are considering is one of a typical procedure for the simulation of a simple car behaviour. This consists of obeying commands from a driver and consequently calculating the acceleration, speed and distance covered by the car and showing their corresponding values on a dashboard display. User commands are of two kinds : driver commands and control commands. The first type includes setting up the acceleration value, setting contact on/off, asking for display of values etc... whilst control commands deal with the reconfiguration of the system e.g. stopping, breaking, repairing or changing any of the display objects. The objects involved and the relationship between them is shown in the application of figure 3.

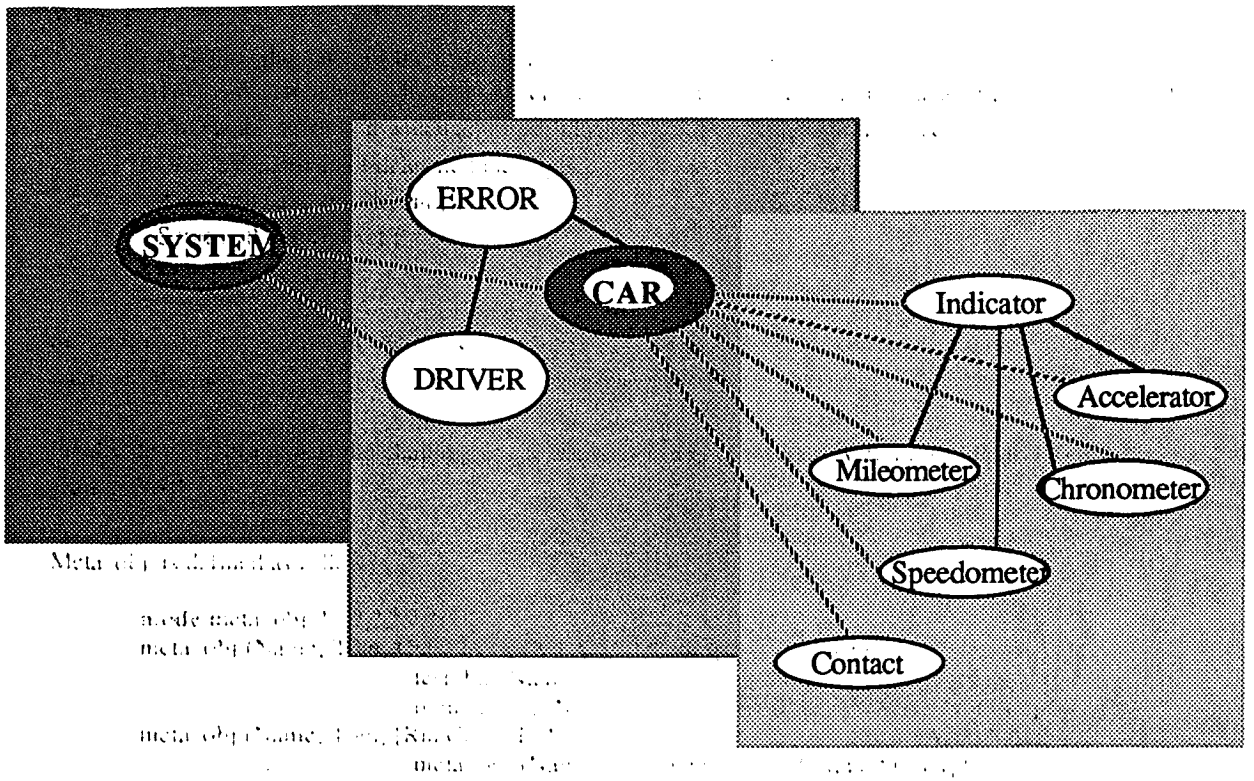

Figure 3 : An Application Network

The important point to note here is the presence of two types of hierarchy : the *part-of hierarchy* and the *is-a hierarchy* which are denoted by dotted lines and solid lines respectively.

3.5.1 The part-of hierarchy

This reflects mainly a 'part-of' decomposition that depends on the nature of objects and that exhibits their relationship with their managers : accelerator, speedometer etc... are parts of object Car. This relationship is also a *control* one since it also reflects the control power a manager can have over objects in its group : suspension, termination, message forwarding, control of message buffering etc...

3.5.2 The is-a hierarchy

This reflects the organisation of objects, not only in terms of their nature but also in terms of their know-how.

In our example above we note the following points :

- an is-a hierarchy comprises only objects of the same abstraction level. This is depicted as a separate plane in the network of figure 3 ;
- all the objects in an is-a hierarchy are under the control of the same object manager. However, a single object manager can be in control of several is-a hierarchies at the same time.

3.5.3 The object activities

As mentioned in section 3.4, an object activity is a stream of messages that the object consumes. These messages include local activities such as that of the alarm clock of figure 2, as well as control activities dealing with the dynamic configuration of the system. We illustrate here the main activities of some of the objects in our example :

object System
activity acquire manager methods, create Driver, create Car, create Error
link Driver to Error, link Car to Error

We see here that an object becomes a manager dynamically. This is interpreted as the acquisition of certain priviliged methods which would enable the creation of Driver and Car. Driver and Car are then linked through the is-a hierarchy to an object error-handler which interprets any unrecognised commands.

Driver interprets commands from the terminal and forwards them in the form of messages to the relevant objects through the part-of hierarchy. Again the object acquires its methods dynamically (section 3.3).

object Driver
activity acquire forwarding method, acquire interaction method,
loop : interact with user, forward corresponding messages.

The activity of object Car is similar to that of System in that it creates its parts objects and the delegation links amongst its objects :

object Car
activity acquire manager methods, create objects Mileometer, Indicator etc...
link Mileometer to Indicator, link Chronometer to Indicator etc...

Finally the activities of Speedometer, Mileometer etc... are more interesting since they include local behavior as well as control messages :

object Speedometer
activity acquire forwarding method,
acquire increment method, acquire speed-value method,
loop: ask Contact for its state, if On update speed

object Mileometer
activity acquire forwarding method,
acquire increment method, acquire distance method,
loop: ask Contact for its state, if On update mileage

The methods acquired are used to interpret messages such as increment, speed-value as well as to forward the ask message to Contact.

3.6 Implementation

We will here illustrate how the above example is realised in PARLOG using our model. As will be seen, this style of programming is not intended to be user-friendly when coded in PARLOG, for this we have implemented a simple preprocessor language called PACTOL (Parlog ACTors/ACTivity Oriented Language) and we are currently implementing a graphical interface for the interactive creation of objects and networks.
Four main points require illustration in PARLOG namely : the object format, the methods format, the setting up of an application , and the specification of activities.

3.6.1 The object format
An object in our model is always created as an instance of a meta-object coupled with an invocation of the objects activity. The latter generates the activity stream 'Acts' to be consumed by the object instance :

```
systemstart <- meta_obj (system, Tom, Cnts, Dels, Acts, St, Scrpt ),
                  St = [initialstate1,state2 ...], Scrpt = [ mm( messagename, methodname ), ...] ) ,
               do_system ( Acts ) .
```

Where :

- system is the name of the instance ;
- Tom is the stream on which an object sends forwarded messages to its manager ;
- Cnts is the stream on which an object manager sends messages to its objects ;
- Dels is the stream of delegated messages ;
- Acts is the activity stream ;
- Scrpt is the Script set ;
- St is the state set.

St is set to the initial state of the object and Scrpt to a set of couples linking a message name to a method that interprets the message.

Finally do_system invokes the chaining of the activity of object System. Activities in PARLOG are described in section 3.6.4.

Meta_obj is defined as follows :

```
mode meta_obj(?,?,?,?,?,?,?).
meta_obj (Name, Tom, Cnts, Dels, [X | Acts], St, Scrpt) <- var (Cnts) :
                        test_buf(Name, Tom, St, NTom, NSt),
                        o_meta_obj (Name, NTom, Cnts, Dels, [X|Acts], NSt, Scrpt).
meta_obj (Name, Tom, [Rm|Cnts], Dels, Acts, St, Scrpt) <- append (Rm, Acts, NActs),
                        meta_obj (Name, Tom, Cnts, Dels, NActs, St, Scrpt).
```

This predicate allows the waiting for messages on both activity (Acts) and control (Cnts) streams. The presence of var(Cnts) in the guard and the append in the second clause means that messages coming on the control stream are given priority over those on the activity stream.

The test_buf simply monitors the number of elements on the streams for buffering. Each time a message is received it updates the value of the object's invariant state element 'messages_waiting' and checks whether the number of messages waiting exceeds the maximum size of the object's message buffer, in which case, it sends a message to the object's manager to suspend the object.

O_meta_obj is defined as follows (the symbol & represents the sequential conjunction) :

```
mode o_meta_obj (?,?,?,?,?,?,?).
o_meta_obj (Name, Tom, Cnts, Dels, [stop|Acts], St, Scrpt) <-
                                find_tail (Tom, Tt) & Tt = [] &
                                find_tail (Dels, Td) & Td = Acts &
                                stop (Name, St, Scrpt) .
o_meta_obj (Name, Tom, Cnts, Dels, [Mess|Acts], St, Scrpt) <-
                        get_met (Mess, Scrpt, Met) :
                        list_to_struct (Met, [Mess, Name, Tom, St, NTom, NSt, Scrpt,
                                                NScrpt, Dels, NDels, Acts, NActs], S) &
                        (call(S) ,
                        meta_obj (Name, NTom, Cnts, NDels, NActs, NSt, NScrpt)) ;
        o_meta_obj (Name, Tom, Cnts, Dels, [Unkw|Acts], St, Scrpt) <-
                        delegate (Name, Unkw, Dels, NDels, St, NSt) ,
                        meta_obj (Name, Tom, Cnts, NDels, Acts, NSt, Scrpt) .
```

This is the predicate that implements the recursion necessary for the realisation of objects (section 2.2). The first clause implements the termination method 'stop' which closes streams and halts the recursion. The second clause searches in the object's script (Scrpt) the method corresponding to the message received, does a metacall to the method and continues the recursion. The third clause implements the delegation method which sends unknown messages on the objects delegation streams Dels and also continues the recursion.

3.6.2 Creation of an Application

The way in which object System is instantiated in the above section in fact occurs only once in an application due to System being the uppermost object in the part-of hierarchy. An object manager requiring the creation of other objects acquires dynamically the special set of methods 'initialize' and 'link' which allow it respectively :

- to create an offspring object in the part-of hierarchy ;
- to link it to other objects in a is-a hierarchy.

The link method is relatively straightforward. In order to link an object 'name' to a series of delegation objects 'name1', 'name2', etc... it simply does a series of merge between the delegation stream 'Dels' of 'name' and each of the activity streams 'Acts' of 'name1', 'name2', etc...

The format of the initialize method is as follows :

```
mode initialize(?,?,?,?,^,^,^,?,^).
initialize(ini(Nam, Inst, Acty, Type, Size),Name Dic, Ls, Acts, NLs, NDic, NActs, Tom, NTom)<-
                update_dict (Tom, Nam, Name, Ntom) &
                update_susplist (Nam, Dic, Ls, Size, NLn, NLs) &
               ( init2 (Nam, NLn, Inst, Acty, Type, Size, Dic, NDic, Tom1),
                merge (Tom1, Acts, NActs)
               ) .
```

where update_dict and update_susplist respectively update the object manager's dictionary and suspension list to contain the newly created object. Init2 subsequently creates the object and the stream link to the object manager which is then merged with the manager's activity stream :

```
mode init2 (?,?,?,?,?,?,?, ^,^).
init2 (Name, NLn, Inst, Acty, Type, Size, dic(Ln, LCi, LCnti), NDic, Tom) <-
        append (Inst, [deleg_set([]), messages_waiting(0), size_max(Size) ], State) &
        list_to_struct ( meta_obj, [Name, Tom, C, D, D_A, State, [mm(add_c, add_c)]] , S) &
        list_to_struct ( Acty, [D_A], Sa) &
        link_cntl_to_dict ( LCi, LCnti, NLn, Type, NDic, Cnti, Cntai) &
       ( call3(S, Stat, Cnti) , call3(Sa, Stat_a, Cntai) ).
```

Init2 here :

- appends the initial instance variables to the invariant state to obtain the object's State ;
- constructs a call to meta_obj as for System above, with State and the invariant acquisition method given by the message/method couple (add_c, add_c) ;
- constructs a call to the object's activity ;
- updates the object's entry in the object dictionary to contain the control flags of the object and its activity.
- finally, does two control metacalls to start the object and its activity. The control metacall (call3) is used to control the suspension/resumption of the object and its activity. This is done in PARLOG by setting the flags Cnti and Cntai to the appropriate values. The returned Status flags Stat and Stat_a are not used.

3.6.3 The method format

A method is always a predicate of a fixed format (number of arguments, modes ...). We illustrate this here with the method for Accelerator that interprets the messages increment and decrement :

```
mode met_acc (?,?,?,?,^,^,?,^,?,^,?,^).
met_acc (inc, Name, Tom, St, Tom, NSt, Scrpt, Scrpt, Dels, Dels, Acts, Acts) <-
                get_state (acceleration(S), St) &
                add (S, 1, NS) &
                put_state (acceleration, [NS], St, NSt).
met_acc (dec, Name, Tom, St, Tom, NSt, Scrpt, Scrpt, Dels, Dels, Acts, Acts) <-
                get_state (acceleration(S), St) &
                sub (S, 1, NS) &
                put_state (acceleration, [NS], St, NSt).
```

The first of the predicate arguments is the name of the message to be interpreted. The other arguments are the same as those described in the context of meta_obj.

The method for increment in Speedometer is slightly more elaborate since it depends on the accelerator_value in Accelerator :

```
mode speedmod (?,?,?,?,^,^,?,^,?,^,?,^).
speedmod (inc, Name, Tom, St, NTom, NSt, Scrpt, Scrpt, Dels, Dels, Acts, Acts) <-
                Tom = [o(accelerator, [acceleration_value(S)]) | C ]  &
                get_state (speed(V), St) &
                add (V, S, NV) &
                put_state (speed, [NV], St, NSt) & NTom =C .
```

Here, the accelerator is forwarded the message acceleration_value(S) using the message o(...) which is sent to the object manager on the Tom stream. The Tom stream is merged with the activity stream of an object manager which, when receiving the o(...) message does the following:

- it looks up in its dictionary for locating the destination object ;
- if the object is directly a part of it, it sends it the message on its Cnts stream,
  otherwise it reforwards the message to a closer object manager.

3.6.4 the object activity

The object activity consists of merging parallel and serial streams into one namely Acts. The activity of Mileometer above is realised as follows :

```
mode do_mil (^) .
do_mil (F) <- F = [add_c( [mm(ext,ext), mm(inc, milmod), mm(position_value, state)] ) | Fs ] ,
                do_mil1 (Fs) .

mode do_mil1 (^) .
do_mil1(F) <- F = [ext(contact, [contact_position(S)]) | F2 ] , do_mil2(S, F2).

mode do_mil2 (?, ^) .
do_mil2 (on, F) <- F = [inc|F2], do_mil1 (F2) .
do_mil2 (off, F) <- do_mil1 (F).
```

Add_c here is the acquire predicate. It specifies that a list of couples (messagename, methodname) is to be added to the object's script. The methods added in this case are the ones specified in section 3.5.3 namely the forwarding method (ext), the increment method(inc, milmod), and the distance method (position_value, state).

Do_mil1 does the forwarding of message contact_position(S), using the method ext, to object contact. Finally, do_mil2, sends the message inc on the activity stream if the state of the contact is on.

5.Conclusions

This paper has described a new object-oriented model based on the concurrent logic programming language PARLOG. This model has the following properties which distinguish it from other models including Vulcan :

i) objects are all instances of a single metaobject in the system. This means that all objects in the system are of the same type and are distinguished only by their behaviour. It also means that each definition of an object in the system does not necessitate the creation of a new recursive predicate, this reduces code proliferation and allows for the tailoring of the behaviour of the metaobject e.g. by replacing the delegate predicate.

ii) control over the behaviour of objects is possible through the special methods used by object managers. This includes controlling the scheduling of objects and the buffering of communicated messages as well as the suspension, resumption and termination of objects ...

iii) it provides for the combination of both a 'part-of' hierarchy as well as an 'is-a' hierarchy with multiple delegation.

iv) all the components of an object are acquired and modified dynamically. This provides a great flexibility in the design of applications of reconfigurable nature.

v) the model introduces the concept of local activities which defines the behaviour of objects.

vi) the power of the concurrent logic model is still apparent in the implementation of the object methods.

Acknowledgments

We are grateful to Najah Naffah and Louis Sauter for supporting this work in the department of advanced studies at Bull under the ESPRIT project#956 COCOS. We would also like to thank Andrew Davison of Imperial College for reading very thoroughly a previous draft of this paper and providing us with many detailed and positive comments.

References

[1] J.S. Conery : *The AND/OR Process Model for Parallel Interpretation of Logic Programs*, PhD thesis, University of California, Irvine (1983).

[2] K. Clark & S. Gregory : "PARLOG : Parallel Programming in Logic", *ACM Trans. on Progr. Languages and Systems* **8** (1), pp. 1-49 (1986).

[3] K. Clark & S. Gregory : "A Relational Language for Parallel Programming", *ACM Conf. on Functional Programming Languages and Computer Architecture*, Portsmouth, pp. 171-178 (1981).

[4] E.Y Shapiro : *A Subset of Concurrent Prolog and its Interpreter*, tech. report TR-003, ICOT, Tokyo (1983).

[5] K. Ueda : *Guarded Horn Clauses*, tech. report TR-103, ICOT, Tokyo (1985).

[6] E.Y Shapiro & A. Takeuchi : "Object Oriented Programming in Concurrent Prolog", *New Generation Computing* **1** (1), pp. 25-48 (1983).

[7] K. Kahn, E.D. Tribble, M.S. Miller & D.G. Bobrow : "Objects in Concurrent Programming Languages", *OOPSLA'86*, Portland, pp. 242-257 (1986).

[8] A. Davison, *"POLKA : A PARLOG Object-Oriented Language"*, internal report, Imperial College, February 1988, London.

[9] M.Ohki, A. Takeuchi, K. Furukawa : " An Object-Oriented Programming Language Based on the Parallel Programming Language KL1", *Proc. Logic Programming Conference*, 1986, London.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique